Design-driven Development of Safety-critical Applications: A Case Study In Avionics

Julien Bruneau, Quentin Enard, Stéphanie Gatti, Emilie Balland, Charles Consel

Design-driven Development of Safety-critical Applications: A Case Study In Avionics

Julien Bruneau, Quentin Enard, Stéphanie Gatti, Emilie Balland, Charles Consel
Thales Airborne Systems, INRIA / University of Bordeaux, France
Email: first.last@inria.fr

***Abstract*—Safety-critical applications have to fulfill stringent requirements, both functional and non-functional. These requirements have to be coherent with each other and must be preserved throughout the software development process. In this context, a design-driven development approach can play a critical role. However existing design-driven development approaches are often general purpose, providing little, if any, conceptual framework to guide the development. The resulting design scope thus becomes largely unpredictable, leading to inconsistencies.**

In this paper, we propose a design-driven methodology that relies on a specific development paradigm. This development paradigm provides a conceptual framework that guides the stakeholders at each development stage. Based on this paradigm, a tool suite provides development support dedicated to each development stage. We demonstrate the benefits of this methodology with a realistic case study in the avionics domain.

I. Introduction

Safety-critical applications are pervasive in domains like railway, avionics and automotive. Development of such applications is constrained by both functional and non-functional requirements, resulting in a strict certification process. The certification consists in demonstrating that the development of the application is compliant with the rules defined by standards such as DO-178B [1]. In particular, the stakeholders have to demonstrate the coherence of the requirements and their conformance at each stage of the development process.

Coherence of the requirements. Functional and non-functional aspects of an application are inherently coupled. For example, dependability mechanisms can potentially deteriorate the overall performance of the application. The coherence of the requirements is particularly critical when the software evolves: even minor modifications to one aspect may tremendously impact the others, leading to unpredicted failures [2].

Conformance across the development stages. To avoid requirements to be partially or totally ignored along the development process, their conformance has to be ensured at each stage. This is generally done manually by tracing their propagation throughout the software development process. In the avionics certification process, traceability (*i.e.,* the ability to trace all the requirements throughout the development process) is mandatory for both functional and non-functional requirements [1].

Certifying a development process requires a variety of activities. In industry, the usual procedures involve holding peer review sessions for coherence verification and writing traceability documents for conformance certification. In this context, design-driven development approaches are paramount because the design drives the development of the application according to the specification of high-level requirements [3]. Such approaches facilitate the traceability of the requirements. However, because most existing approaches are general purpose, they provide no conceptual framework for guiding the development, potentially leading to inconsistencies. This situation calls for an integrated development process that provides such conceptual framework, allowing to automate as much verification as possible towards software certification.

In this paper, we propose an integrated design-driven development methodology for safety-critical applications. This methodology relies on a development paradigm and a tool suite. The paradigm provides a design framework to express both functional and non-functional requirements. As demonstrated by Shaw, the use of a specific paradigm allows a more disciplined engineering process [4]. In particular, this approach enables to focus on domain-specific problems and safety issues in the early development stages. To support this development paradigm, we rely on an existing tool suite that provides support for both functional and non-functional development aspects [5]–[7]. The paradigm is enforced by a unique design language, ensuring the coherence of functional and non-functional specifications. From the design, a compiler generates a dedicated programming and testing support; they both contribute to enforcing the conformance with the specifications at each development stage.

Contributions

So far, the benefits of our design-driven development approach has been demonstrated in isolation, that is, with respect to individual functional and non-functional aspects [5]–[7]. Coherence and conformance remain to be addressed in an integrated manner, and validated in the context of realistic applications. This paper takes up these challenges, making the following contributions.

An automated approach to preserving coherence. We propose an approach that relies on a dedicated paradigm to design both functional and non-functional specifications of safety-critical applications. This paradigm takes the form of a design language named DiaSpec [8].

An automated approach to ensuring conformance. A DiaSpec specification is used to generate a dedicated programming framework and a related testing framework. By construction, we ensure the conformance of the requirements between each stage of the software development process, achieving the traceability of the requirements throughout the development process.

An integrated approach. While the coherence of functional and non-functional specifications is ensured by the design framework, the conformance between the development stages is realized by guiding the development process, generating traceable programming and testing support. This integrated approach automates the verification of coherence, and traceability throughout the development process, reducing the amount of efforts required for the development of safety-critical applications.

Validation in avionics. We have validated our development approach by developing flight guidance applications for avionics and drone systems. We deployed the avionics flight guidance application on a realistic flight simulator [9], and the drone flight guidance application on a commercial vehicule [10].

II. Case Study: Flight Guidance

This section presents the functional and non-functional requirements of the flight guidance application of a plane. This safety-critical application is the case study used throughout this paper.

A. Functional Requirements

The flight guidance application is in charge of the plane navigation and is under the supervision of the pilot. For example, the pilot can directly specify parameters during the flight (*e.g.,* the altitude) or define a flight plan that is automatically followed.

Each parameter is handled by a specific navigation mode (*e.g.,* altitude mode, heading mode). Once a mode is selected by the pilot, the flight guidance application is in charge of operating the ailerons and the elevators to reach the target position [11]. For example, if the pilot specifies a heading to follow, the application compares it to the current heading, sensed by devices such as the Inertial Reference Unit, and maneuvers the ailerons accordingly. In the avionics domain, each of these modes is generally associated to a *functional chain* [12], representing a chain of computations, from sensors to actuators.

[1]The Air Data Inertial Reference Unit (ADIRU) is a component that supplies air data and inertial reference.

Requirement	Description
Req1. The execution time of the heading mode must not exceed 650 ms.	It ensures that the frequency of computation of the heading mode does not lead to an unexpected behavior of the plane.
Req2. The freshness of the navigation data used by the application must be less than 200 ms.	The use of an outdated navigation data can lead to erroneous decisions.
Req3. The ADIRU [1] must be replicated twice to tolerate at least one crash failure.	It ensures the availability of navigation data, despite the loss of a sensor.
Req4. Any malfunctioning or lost sensor must be signaled to the pilot, with identification of the probable cause.	Decisions taken by the pilot are based on information about the sensors' state.
Req5. A navigation mode must be deactivated if the required data is unavailable.	Without appropriate data, a navigation mode cannot safely control the plane.
Req6. Information related to the activation/deactivation of navigation modes must be logged.	Application monitoring is used for maintenance.

Figure 1: Extract of the safety requirements of the flight guidance application.

B. Non-Functional Requirements

To identify hazardous situations, safety analyses are conducted [13], resulting in safety requirements. Grunske presents safety requirements as a formal description of a hazard, combined with the tolerated probability of this hazard [14]. Then, he classifies hazard description in three categories: (1) the system is not available, (2) the system generates an incorrect output, and (3) the system misses a hard deadline. The tolerated probability of the hazard, depending on its effect on the passengers and the flight crew, leads to the attribution of a Design Assurance Level (DAL) [1]. For example, the flight guidance application has the highest DAL (A level) as a failure may cause a crash. Assigning a DAL to an application results in defining a set of non-functional requirements for this application to avoid or minimize the identified hazardous situations [15]. Figure 1 gives a representative subset of the non-functional requirements for the flight guidance application, as defined by domain experts. These requirements define constraints to several non-functional aspects of the application. The **Req1** and **Req2** requirements define constraints on the application performance. The **Req3**, **Req4** and **Req5** requirements define constraints on its reliability. The **Req6** requirement corresponds to maintenance constraints.

In practice, there exist dependencies between functional and non-functional requirements. The coherence and the traceability of these requirements thus become critical, strongly suggesting a global approach to software design and development.

III. Methodology

Our development methodology relies on both a development paradigm and a tool suite. The paradigm provides a *conceptual framework* that guides the stakeholders at each development stage. Based on this paradigm, the tool suite provides an *integrated development environment* that provides specific support for each development stage.

We first present the development paradigm, abstracting over tool support. Then, we examine how our tool suite supports the development paradigm.

A. A paradigm-based methodology

Our methodology relies on the Sense/Compute/Control (SCC) paradigm [7]. This paradigm orginates from the *sense/compute/control* architectural pattern, promoted by Taylor *et al.* [16]. This paradigm applies to applications that interact with an external environment. Such applications are typical of domains such as building automation, robotics, and autonomic computing.

As depicted in Figure 2, the underlying design pattern consists of *context components* fueled by sensing *entities*. These components refine (aggregate and interpret) the information given by the sensors. These refined data are then passed to *controller components* that trigger actions on entities. For example, in the heading mode, the flight guidance application senses the environment to acquire heading data (*i.e.,* magnetic and/or gyroscopic heading measurements). Then, the application uses this raw data to compute information needed to control the plane heading (*i.e.,* the differential angle to apply on ailerons to reach a target heading).

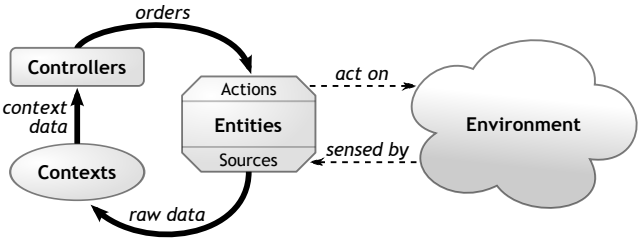

Figure 2: The SCC paradigm

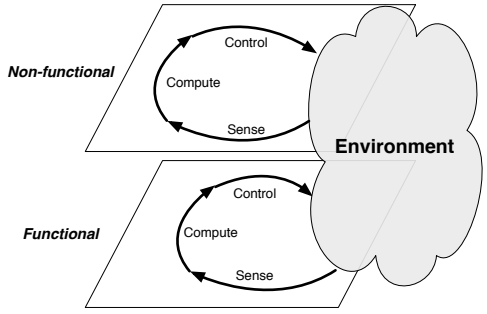

Figure 3: Layered view of the SCC paradigm

Like a programming paradigm, the SCC paradigm provides concepts and abstractions to solve a software engineering problem. However, these concepts and abstractions are dedicated to a design style, raising the level of abstraction above programming. Because of its dedicated nature, such a development paradigm allows a more disciplined engineering process, as advocated by Shaw [4], [17].

As shown in Figure 3, the SCC paradigm can be used to describe both the functional and non-functional aspects of the application, using several SCC layers. For example, the **Req5** requirement entails the deactivation of the navigation modes that rely on defective sensors. In this case, the functional part of the flight guidance application becomes the environment for a monitoring SCC loop. This SCC loop senses the state of the navigation sensors and computes refined information to determine whether data sources have become unavailable. If so, the control consists in reconfiguring the application to deactivate the dependent modes.

To benefit from the SCC paradigm throughout the development process, we propose the following methodology:

Step 1: Design.

- *Step 1.1: Taxonomy.* The domain expert describes classes of *entities*, modeling the environment. An entity is defined as a set of data sources and actuating capabilities, abstracting over devices, whether hardware or software.
- *Step 1.2: Application Data Flow.* The designer decomposes an application into functional components with respect to the SCC style. This decomposition takes the form of a data-flow directed graph, where a node is an SCC component (entity, context and controller) and the edges indicate data exchange between the components.
- *Step 1.3: Application Control Flow.* The designer specifies what interaction a given component can perform, expressing in high-level terms control-flow constraints dedicated to the SCC paradigm [7]. These control-flow constraints are used to enrich the data-flow graph by refining the edges to make the interaction types explicit. These interaction specifications allow the designer to verify at design time a range of properties, such as the failure impact of a sensor.
- *Step 1.4: Non-Functional Specification.* The functional design obtained in the previous steps is refined with non-functional requirements. For example, the taxonomy is enriched with information about entity failures, and the component graph is leveraged to express timing constraints. Moreover, non-functional treatments are specified alongside the application logic, introducing a separation of concerns at the

design level. For example, the deactivation of navigation modes is a non-functional treatment expressed as a dedicated SCC specification, consuming non-functional inputs (*e.g.,* errors). We rely on the SCC paradigm to provide a uniform approach to integrating functional and non-functional treatments.

Step 2: Implementation.
The design obtained in Step 1 systematically guides and constrains the implementation process. The SCC paradigm separates the functional and non-functional specifications, allowing their implementation to be performed independently. For example, safety experts can concentrate their development efforts on critical subsystems exposed at the design level (*e.g.,* monitoring and reconfiguration), instead of inspecting the code for such operations as exception handling. This separation of concerns is essential to open safety-critical domains, such as avionics, to modular software engineering where a platform is assembled from software components involving a range of stakeholders.

Step 3: Testing.
Like the earlier development stages, testing relies on the SCC paradigm.

- *Step 3.1: Early Validation of the Specifications.* Given the design of an application, verifications are performed to ensure the conformance between the specification and the requirements. For example, the QoS specification allows to validate the performance for a specific deployment configuration, using prediction tool and deterministic QoS characteristics of the execution platform (*e.g.,* the deterministic performance of the AFDX network [18]).
- *Step 3.2: Implementation Testing.* The implementation of each SCC layer can be tested independently. For example, the functional aspect of the application can be tested using a simulated external environment. The taxonomy definition allows to validate the functional implementation using mock-up entities that rely on the simulated environment, without any impact on the rest of the application.

Our methodology provides a paradigm that guides the stakeholders all along the development. They have to express the functional and non-functional aspects of the application according to the paradigm rules.

Although our paradigm-based methodology introduces a disciplined and systematic development process, the quality of the resulting software system is still unknown because the process relies on the developers' rigor to apply our guidelines.

B. A tool-based methodology

To change the contemplative nature of our approach, we introduce a design language and a suite of tools that guide, assist and verify the development process. Our design language provides the developers with syntax and semantic concepts dedicated to SCC applications [8]. The tool-based version of our methodology revolves around the functional and non-functional specification of an application; it is named DiaSuite[2] and is depicted in Figure 4.

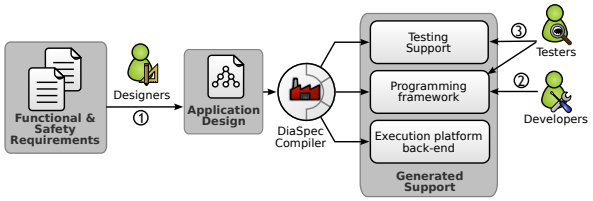

Figure 4: The DiaSuite tool-based methodology

1) The design stage: This stage is now concretized with a design language, named DiaSpec, dedicated to specifying SCC applications. DiaSpec provides the designer with syntactic constructs to express SCC concepts in response to functional and non-functional requirements. These constructs allow to declare an entity taxonomy as well as the functional and non-functional specification of an application (stage ①). More precisely, DiaSpec includes design constructs to declare *interaction contracts* between SCC components, error handling aspects and QoS constraints [5]–[7]. Functional and non-functional declarations are uniformely integrated in a design language, ensuring their coherence. For example, DiaSpec includes error handling declarations; when an entity is declared as raising a type of error, the dependent SCC components are required to declare a type of treatment.

2) The implementation stage: Leveraging a DiaSpec description, the DiaSuite compiler generates a dedicated programming framework that provides high-level programming support (stage ②), guiding the implementation stage. Moreover, the code resulting from the data-flow and control-flow declarations of the DiaSpec description ensures the conformance between the implementation and the design [8], whether functional or non-functional. For example, declaring an SCC component as treating an error type forces the implementer to write handling code.

3) The testing stage: As the taxonomy abstracts over the heterogeneity of hardware and software entities, the application can be simulated for the early validation of functional and safety requirements. In the context of the avionics domain, we validated the behavior of our flight guidance application in a simulated environment, leveraging both the DiaSuite-generated programming framework and an existing, realistic flight simulator, namely FlightGear [9] (stage ③). Moreover, the tool suite provides testing support to customize the non-functional

[2]http://diasuite.inria.fr

properties of the application during the simulation, allowing the verification of the application behavior in exceptional conditions.

Let us now present in detail each stage of our tool-based methodology, illustrated with our case study of flight guidance.

IV. Design

This section presents the design stage of our tool-based methodology. This stage is illustrated with the design of the heading mode of the flight guidance application, expressed in DiaSpec [3].

A. Taxonomy

We first identify the *entities* that interact with the environment and are required to control the heading. The plane heading is provided by Inertial Reference Units (IRUs). These units encapsulate accelerometers, gyroscopes, and GPS sensors, and provide navigation data. To allow the pilot to set a heading, we define a user-interaction entity, namely Man-Machine Interface (MMI). Finally, controlling the plane heading requires to act on the plane ailerons. These entities are specified in the taxonomy presented in Figure 5. Entities are declared using the **device** keyword. The `IRU` entity senses the position and the heading of the plane from the environment as indicated by the **source** keyword. The `NavigationMMI` entity abstracts over the pilot interaction and directly provides the target heading. The `Aileron` entity provides the `Control` interface to the application as indicated by the **action** keyword.

The high-level nature of the taxonomy definitions facilitates the integration of Commercial Off-The-Shelf (COTS) components: any implementation complying with an entity declaration can be used by an application.

B. Application Data Flow

Using the taxonomy, the designer then specifies the application data flow by declaring *context* and *controller* components. Figure 8 presents the design fragment of the flight guidance application related to the heading mode. From bottom to top, the process can be summarized as follows. The `IntermediateHeading` context component computes an intermediate heading from the current plane heading and the target heading. Given this heading and the current plane roll (*i.e.,* its rotation on the longitudinal axis), the `TargetRoll` context component computes a target roll. This target roll is used by `AileronController` to control the ailerons and reach the target heading.

The SCC paradigm facilitates the evolution. For example, as depicted in Figure 8, the `IntermediateHeading` context component abstracts over the computation of the target heading. Indeed, it can be computed either from `targetHeading` directly provided by `NavigationMMI` or from the target heading computed by `HeadingToWaypoint`. The `HeadingToWaypoint` context component computes a target heading to reach the next waypoint provided by the route manager.

[3]The full DiaSpec specification can be found at http://diasuite.inria.fr/avionics/51

```
device IRU {
  source heading as Float [frequency 200ms];
  source position as Coordinates;
  ...
  action Deactivate;
  raises FailureException;
}
device NavigationMMI {
  source targetHeading as Float;
  ...
  action DisableMode;
}
action Control{
  incline(targetRoll as Float);
}
device Aileron {
  action Control;
}
```

Figure 5: Extract of the flight guidance taxonomy

```
context IntermediateHeading as Float {
  source heading from IRU;
  source targetHeading from NavigationMMI;
  context HeadingToWaypoint;
  interaction {
    when provided heading from IRU;
    get targetHeading from NavigationMMI,
      HeadingToWaypoint;
    always publish;
  }
}
```

Figure 6: Specification of `IntermediateHeading`

```
context IntermediateHeading as Float {
  source heading from IRU;
  source targetHeading from NavigationMMI;
  context HeadingToWaypoint;
  interaction {
    when provided heading from IRU;
    get targetHeading from NavigationMMI in 100ms
        [mandatory catch],
      HeadingToWaypoint;
    always publish;
  }
}
```

Figure 7: Refinement of `IntermediateHeading`

C. Application Control Flow

In DiaSpec, the control flow of the application components is expressed with interaction contracts [7]. The specification of an SCC component with its interaction contract is illustrated in Figure 6. This DiaSpec fragment declares the `IntermediateHeading` context component as producing an intermediate heading of a `Float` type from three inputs: two are entities, declared with the **source** keyword, and one is a context component, declared with the **context** keyword. The control flow of

this process is specified by the interaction contract introduced by the **interaction** clause. It declares that, when `IntermediateHeading` receives a `heading` information from the `IRU` entity, it may access the `targetHeading` value provided by the `NavigationMMI` entity and the information provided by the `HeadingToWaypoint` context. The **always publish** clause specifies that the context systematically publishes a value once it receives a `heading` information. Alternatively, a context component can be declared as either maybe or never publishing a result, by including the **maybe publish** or **no publish** clause, respectively.

D. Non-functional Specification

The non-functional specification refines the design of the application produced by steps (1.1) to (1.3) of our methodology. Moreover, additional SCC layers may be specified to deal with the non-functional aspects of the application (*e.g.,* reconfiguration).

1) Taxonomy: The designer defines the potential failures of entities. This is realized by declaring the errors an entity can raise. These errors are associated with an entity (a source or an action). In the specification of `IRU` presented in Figure 5, this entity is declared as raising an error of type `FailureException`. Handling an error may require declaring treatment alongside the functional specification, as described in Section IV-D3.

The designer may also refine the taxonomy with timing constraints. Examples include the frequency at which an entity source produces data, or the response time of data access. Figure 5 illustrates the specification of the frequency of data production. In this example, it is specified that the `IRU` entity produces the `heading` information with a frequency of 200 milliseconds. This constraint is derived from the safety requirement **Req2** presented in Figure 1.

2) Application design: The application design (steps (1.2) and (1.3)) is also refined to integrate non-functional specifications. For example, the refinement of the `IntermediateHeading` context specification is illustrated in Figure 7. In the interaction contract of `IntermediateHeading`, the response time of `NavigationMMI` has to be at most 100 ms. The `[mandatory catch]` annotation indicates that the `IntermediateHeading` context must compensate the errors when accessing `targetHeading` data. In contrast, the `[skipped catch]` annotation indicates that a context is not allowed to handle the errors. In Figure 8, these non-functional specifications are indicated in brackets.

3) Non-functional SCC layers: Alongside the application logic, non-functional treatments can be specified in DiaSpec as depicted by the right part of Figure 8. In the avionics domain, these treatments typically involve monitoring the application and triggering reconfigurations, as required by **Req4** and **Req5** and **Req6** in Figure 1. Specifically, these non-functional treatments allow to (1) inform the pilot in case of a device failure or unavailable data, (2) deactivate the modes that depend of unavailable data, and (3) log information for maintenance purposes.

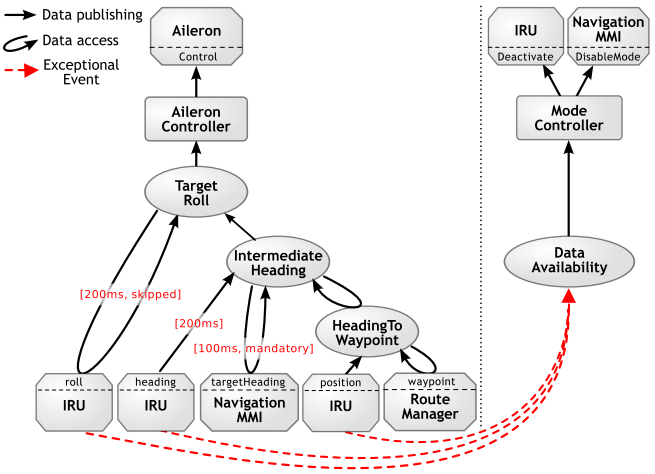

Figure 8: Extract of the flight guidance application design

Non-functional treatments are specified with respect to non-functional information defined in the taxonomy and the application design. For instance, errors raised by entities or the violation of timing constraints are used as sources of information for the non-functional treatments. In Figure 8, the availability of `IRU` data is checked through the `DataAvailability` context component and is then used by the `ModeController` component to enable/disable navigation modes and inform the pilot.

V. Implementation

When developing safety-critical applications, a key goal is to preserve the functional and non-functional requirements throughout the development process. To do so, the DiaSuite approach relies on a compiler that generates a dedicated programming framework from a DiaSpec design. This generative approach ensures the conformance between the design and the implementation stages, while offering high-level programming support to the developers.

A. Functional Programming Support

As depicted in Figure 4, the compiler takes as input the DiaSpec specification of the application and generates a dedicated Java programming framework that ensures the conformance between the design and the implementation [8].

For example, Figure 9 shows the abstract class generated from the specification of the `IntermediateHeading` context component. This abstract class guides the developer by providing high-level operations for entity binding and component interactions. Additionally, our generation strategy of an abstract class leverages the Java language and its type system to enforce the declared interaction

contracts. Concretely, when extending the `AbstractIntermediateHeading` abstract class, the developer is required to implement the `onHeadingFromIRU` abstract method to receive a value published by this device. In addition to this value, this method is passed support objects to request data from a device (`binding`) and a context component (`getContext`).

```
public abstract class
  AbstractIntermediateHeading {

  public abstract Float onHeadingFromIRU(
    Float heading,
    Binding binding,
    GetContext getContext);
  ...
}
```

Figure 9: Extract of the `AbstractIntermediateHeading` class

```
public class IntermediateHeading extends
    AbstractIntermediateHeading {

  public Float onHeadingFromIRU(
      Float heading, Binding binding, GetContext
          getContext) {
    NavigationMMI mmi = binding.navigationMMI();
    Float heading = mmi.getTargetHeading(
      new TargetHeadingContinuation() {
        public Float onError() { return
            DEFAULT_VALUE; }
      }
    );
  }
  ...
}
```

Figure 10: Extract of the `IntermediateHeading` context implementation

The *control inversion* principle is uniformly applied to an SCC-generated programming framework to guarantee that the interaction between the components are conform to the design. Specifically, the abstract methods to be implemented by the developer are only called by the framework, ensuring that a DiaSpec software system is compliant with its DiaSpec design.

B. Non-Functional Programming Support

The non-functional specifications are preserved throughout the implementation stage by generating a dedicated programming framework [5], [6]. For example, the `IRU` entity was declared in the taxonomy (Figure 5) as raising `FailureException` errors. Consequently, a specific method is generated in the corresponding entity abstract class to allow error signaling to be introduced by the developer when implementing an instance of this entity [5]. Another example is the **mandatory catch** declaration in the `IntermediateHeading` interaction contract presented in Figure 7. This declaration imposes the `IntermediateHeading` implementation to handle potential errors when requesting the `targetHeading` data from `NavigationMMI`. As shown in Figure 10, this mandatory error handling is enforced by introducing a continuation parameter in the method supplied to the developer to request the `targetHeading` data (*i.e.,* `getTargetHeading`). This continuation provides a default value in case of an error.

Timing constraints specified at design time are also preserved in the generated programming framework. These constraints are automatically monitored in the generated programming framework. For instance, it monitors the time spent by the `IntermediateHeading` context to retrieve the `targetHeading` data. If this time is greater than 100 ms (as specified in Figure 7), an error is automatically raised by the framework. This approach allows the developer to focus on the code to handle the violation of timing constraints.

Non-functional treatments are handled independently from the application logic. This separation of concerns allows a developer to focus on a specific non-functional aspect. For example, the developer of the `DataAvailability` context can concentrate on implementing algorithms to detect data availability. Because of the programming framework support, the developer does not need to mix detection and error handling code with the application logic, keeping separate the functional and non-functional treatments.

VI. Testing

This section presents the support provided by our tool-based methodology for the testing stage. First, we illustrate the early validation of the application, leveraging design-time verifications. Then, we show the simulation support provided by our tool suite for testing the implementation.

A. Early Validation

Because the DiaSpec design language makes flow information explicit, a range of properties can be checked at design time.

In previous works, we have shown how to verify properties such as *interaction invariants* using model-checking techniques [7]. For example, in our flight guidance application, we can verify that the failure of an IRU always results in a warning message on the pilot's display unit. As this invariant results directly from the requirement **Req4** (Section II-B), its validation at design time ensures the conformance of the specification with this requirement. Moreover, the generation of the programming framework ensures that this property is preserved at the implementation level [7].

Early validation also applies to time-related requirements. From the timing constraints defined in the design, we can generate a set of equations [6]. For example, the execution time of the heading mode can be decomposed into: (1) a communication time between context and controller components, defined by the type

of communication and the media used (*e.g.,* AFDX is a commonly used network in avionics with a deterministic communication time [18]), (2) a computation time of context and controller components, and (3) a time to acquire data from the sensing devices. As established by **Req1**, the execution time of the heading mode must not exceed 650 ms. By injecting timing constraints such as distributed systems technologies, platform and hardware characteristics, we can check whether a given deployment configuration is in conformance with the QoS specifications.

B. Simulation Support

The implementation of each SCC part can be tested independently. For example, the functional aspect of the application can be tested using a simulated external environment. The taxonomy definition allows to validate the functional implementation using mock-up entities that rely on the simulated environment, without any impact on the rest of the application.

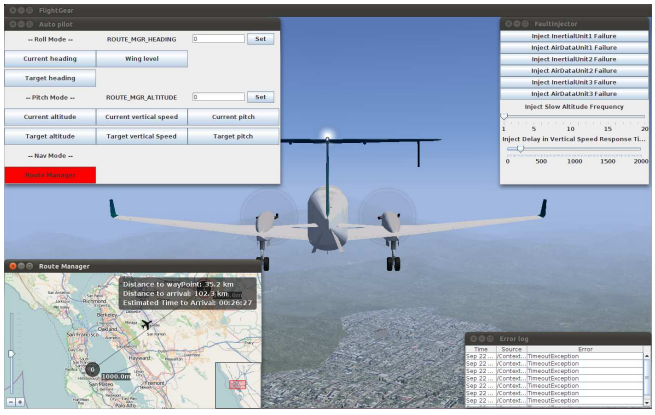

Figure 11: Screenshot of a simulated flight

```
public class SimulatedIRU extends AbstractIRU
                    implements SimulatorListener {

  public SimulatedIRU(FGModel model) {
    model.addListener(this);
  }

  public void simulationUpdated(FGModel model) {
    publishPosition(model.getInertialPosition());
  }

}
```

Figure 12: Extract of the simulated `IRU` class

For example, in avionics, it is required to verify the behavior of the application in specific environmental conditions. Because some scenarios are difficult to create (*e.g.,* extreme flight conditions), we provide a testing support that relies on a flight simulator, namely FlightGear [9], to simulate the external environment.

Using a Java library that interfaces with FlightGear, the testers can easily implement simulated versions of entities. Figure 12 presents an extract of the implementation of a simulated IRU.

The `SimulatedIRU` entity is implemented by inheriting the `AbstractIRU` class provided by the programming framework. To interact with the simulated environment, the entity implements the `SimulatorListener` interface. This interface defines a method named `simulationUpdated`, which is called periodically by the simulation library. The `model` parameter allows to read/write the current state of the FlightGear simulator. In Figure 12, the position of the plane is published by calling the `publishPosition` method of the `AbstractIRU` class.

Once the simulated entities are implemented, the flight guidance application is tested by controlling a simulated plane within FlightGear. Figure 11 presents a screenshot of our testing environment. In the main window, the FlightGear simulator allows to control and visualize the simulated plane. In the top-left corner, the autopilot interface allows testers to select a navigation mode. In this case, the "Route Manager" mode is selected to follow the flight plan defined via the map displayed in the bottom-left corner. This simulated environment is also useful to the non-functional SCC layers. Instrument failures can be directly simulated using FlightGear. We also provide a simple simulation support to inject errors from the simulated entities as illustrated by the `FaultInjector` window in the top-right corner. Then, the window in the bottom-right of the screenshot displays the errors monitored by the application.

Finally, it is required to realize integration testing on a test bench to ensure that the application behaves correctly for a specific deployment configuration. An advantage of our simulation support is that simulated and real entities can be combined in a hybrid environment. Indeed, as both real and simulated versions of an entity extend the same abstract class, the nature of an entity has no impact for the rest of the application. Deploying an application on a test bench is a daunting task that has to be repeated each time an error is detected. Testing by simulation the most unreliable components prior to a test bench may avoid unnecessary deployments.

VII. Assessment

We now outline the benefits of our methodology, focusing on the coherence and conformance requirements. These benefits are illustrated by the flight guidance case study. Then, we mention the benefits of our approach for reuse by briefly describing the development of a flight guidance application for a commercial drone.

A. Coherence

The DiaSpec language has been designed to enable the coherence checking of a specification. In the spirit of a type checker, the DiaSpec compiler checks that the design respects the SCC paradigm. For example, a context component cannot act on an entity; as well, the activation

condition of an interaction contract cannot be a context that never publishes.

Moreover, uniformly integrating functional and non-functional aspects in a design language prevents most inconsistencies that occur when these aspects belong to independent views (*e.g.,* the collection of UML diagrams). For example, the coherence between timing constraints can be statically checked as they directly refine the interaction contracts describing the control flow [6]. If the designer declares a component as requiring a specific data freshness, compile-time verifications ensure that this component can only be fueled by entities providing data in a shorter time.

The coherence is preserved at the implementation level thanks to the generated programming framework. A component can only communicate with the other components in conformance with its interaction contracts, ensuring communication integrity [19]. Similarly, the generated support for the non-functional specification preserves the coherence. For example, the support generated for error handling, such as in the `DataAvailability` context component, prevents developers from implementing ad-hoc code for the propagation of errors between components. This separation of concerns in the programming framework allows developers to focus on their area of expertise without introducing unintentional inconsistencies.

B. Conformance

The generation of a programming framework relies on the control inversion principle presented in Section V, enforcing the conformance with the design. Indeed, the developers have to extend the abstract classes of the framework to implement the application logic and thus cannot introduce inconsistencies with respect to the design. For example, the interaction contract of `IntermediateHeading` (see Figure 7) results in the generation of an abstract method (see Figure 9) that will be automatically called each time the `IRU` entity publishes heading data. As the programming support for getting `targetHeading` data is only provided through parameters of this method, developers can only access it once the method is called by the framework [7]. Similarly, we have shown how the non-functional specifications were preserved by generating dedicated mechanisms (*e.g.,* continuations that enforce error handling code to be provided by the developer).

As the generated programming framework ensures the conformance with the design throughout the software development, we claim that such generative approach could greatly facilitate the traceability burden of the certification process.

C. Design-driven Reuse

The development paradigm proposed in this paper promotes reuse. For example, a taxonomy abstracts over the variability of the concrete entities; thus, the same taxonomy can be reused for all the applications that interact with a similar external environment. More generally, the decomposition of the SCC pattern in layers facilitates the reuse of components. To evaluate the benefits of our approach in terms of reuse, we propose to adapt the avionics flight guidance application to a drone platform. We consider that the drone is composed of an accelerometer, two gyrometers and a front-facing camera. This configuration is standard in commercial drones such as the Parrot A.R. Drone [10].

In commercial drones, the control is generally realized by the user through a smartphone. The goal of our flight guidance application is to make the drone autonomous by following a flight plan similar to the one in avionics. In the resulting design, the avionics taxonomy can be partially reused. For example, the entities related to the flight plan (*e.g.,* the `RouteManager` entity) and the entities used for logging and displaying information have been reused. Most of the navigation data provided by the drone are easily convertible into high-level information usable by the avionics application, allowing to reuse most of the context and controller components. For example, the heading of the drone can be calculated from the angle of rotation around the vertical axis. Concerning the non-functional aspects, the SCC layer dedicated to the control of the failures has been enriched with drone-specific constraints such as battery handling. In total, the drone application consists of 5 reused entities out of 7, and 18 reused SCC components out 30. This application has been successfully deployed on the Parrot's A.R. Drone [10]. The full DiaSpec specification and a video demonstrating this application are available online [4].

VIII. Related Work

Several design-driven development approaches are dedicated to applications with stringent non-functional requirements.

In the domain of architecture description languages, the Architecture Analysis & Design Language (AADL) is a standard dedicated to real-time embedded systems [20]. AADL provides language constructs for the specification of software systems (*e.g.,* component, port) and their deployment on execution platforms (*e.g.,* thread, process, memory). Using AADL, designers specify non-functional aspects by adding properties on language constructs (*e.g.,* the period of a thread) or using language extensions such as the Error Model Annex [5]. The software design concepts of AADL are still rather general purpose and gives little guidance to the designer. At the expense of generality, our approach makes explicit domain-specific concepts in the design specification of a software system, namely sensors, contexts, controllers, actuators. This approach enables

[4] http://diasuite.inria.fr/avionics/52

[5] The Error Model Annex is a standardized AADL extension for the description of errors [21].

further development support for design, programming, verification and deployment.

As AADL is a standard, a lot of research has been devoted to provide it with analysis and development tool support. For example, Dissaux *et al.* present performance analysis of real-time architectures [22]. They propose a set of AADL design patterns to model real-time issues, such as thread synchronization. For each pattern, they list a set of performance criteria (*e.g.,* the bounds on a thread waiting time due to access data) that can be checked with a performance analysis tool [23]. In contrast, our design framework allows higher-level specifications and analysis of timing constraints, abstracting over real-time systems issues [6]. As AADL mainly focuses on deployment concerns, it is complementary to our approach and could be used for the deployment specification and analysis of DiaSpec applications. While most ADLs provide little or no implementation support, the Ocarina environment allows the generation of programming support dedicated to an AADL description [24]. However, this programming support consists of glue code for a real-time middleware and does not guide nor constrain the application logic implementation.

In model-driven engineering, several approaches focus on safety-critical applications. For example, Burmester *et al.* propose a development approach dedicated to mechatronic systems [25]. This approach is based on a domain-specific extension of UML for real-time systems. To allow the formal verification of a whole mechatronic system, the authors propose to develop a library of coordination patterns that define specific component roles, their interactions and real-time constraints. Then, the components of the application are built using this library of patterns by specifying their roles and additional behavior details. The approach comprises tool support for the specification, verification and source code synthesis as a plug-in for the Fujaba tool suite. The use of coordination patterns can be seen as a paradigm that guides the design of mechatronic systems. Thus, this approach is similar to ours, while focusing on a different application domain. Another example of model-driven approaches for safety-critical application is the work of Faugere *et al.* where a UML profile dedicated to real-time systems, named MARTE [26], is used to specify and verify applications. MARTE is viewed as a high-level specification language to allocate software applications to hardware resources and verify real-time properties, such as timeliness and schedulability. Then, a MARTE design is translated into an AADL specification to take advantage of the AADL tool support. MARTE leverages AADL to provide specification support for the deployment stage; it is complementary to our work.

SCADE (Safety Critical Application Development Environment) is the development methodology that is the most similar to ours [27]. SCADE is based on a synchronous language and relies on hierarchical state machines for the specification of safety-critical applications. Non-functional aspects are specified using state machines and their coherence is verified at design time. The synchronous paradigm ensures by construction the determinism of a specification, and thus eases these verifications. The approach abstracts over physical time allowing real-time properties to be verified at the code level. Our design methodology is similar to this approach but lifts constraints inherent to the determinism of the specification for promoting reuse. SCADE could be used to specify more precisely the internal behavior of critical DiaSpec components. The SCADE tools preserve determinism from the specification to the implementation via code generation. Indeed, these tools have been certified to ensure the conformance between the design and a generated implementation. While not certified, the DiaSpec compiler generates a programming framework and leverages the type checker of the host language to ensure such conformance. Instead of proving the correctness of the DiaSpec compiler, one solution could be to generate certifiable code similarly to the approach proposed by Denney *et al.*. Their approach is based on the automatic annotation of generated code with safety proofs under the form of pre/post conditions and invariants [28].

IX. Conclusion and future works

In this paper, we have presented a design-driven methodology for the development of safety-critical applications. This methodology relies on a specific development paradigm, the *Sense-Compute-Control* paradigm, that provides a conceptual framework guiding the stakeholders at each stage. In contrast with general-purpose design driven approaches, such as UML, the SCC pattern guides the development rigorously for both functional and non-functional specifications. Based on this paradigm, DiaSuite provides specific support for each development stage while guaranteeing the conformance with the specifications. We have demonstrated the benefits of our methodology by developing a flight guidance application that has been deployed in a simulated environment and in a commercial drone platform.

We are currently working on the specification of fault tolerance strategies at the design level to generate more support for error handling. Another direction concerns the deployment stage. We would like to rely on existing avionics deployment technologies to provide deployment support guided by the design.

References

[1] "DO-178B, Software Considerations in Airborne Systems and Equipment Certification (RTCA, Inc.)," 1992. [Online]. Available: http://www.rtca.org/

[2] B. Littlewood and L. Strigini, "Software Reliability and Dependability: a Roadmap," in *ICSE'00: Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM, 2000, pp. 175–188.

[3] M. Volter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-driven Software Development: Technology, Engineering, Management*. John Wiley and Sons Ltd, 2006.

[4] M. Shaw, "Beyond Objects: A Software Design Paradigm Based on Process Control," *SIGSOFT Software Engineering Notes*, vol. 20, pp. 27–38, January 1995.

[5] J. Mercadal, Q. Enard, C. Consel, and N. Loriant, "A Domain-Specific Approach to Architecturing Error Handling in Pervasive Computing," in *OOPSLA'10: Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*, Reno United States, 10 2010.

[6] S. Gatti, E. Balland, and C. Consel, "A Step-wise Approach for Integrating QoS throughout Software Development," in *FASE'11: Proceedings of the 14th European Conference on Fundamental Approaches to Software Engineering*, Sarrebruck Germany, 03 2011.

[7] D. Cassou, E. Balland, C. Consel, and J. Lawall, "Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications," in *ICSE'11: Proceedings of the 33rd International Conference on Software Engineering*. Honolulu United States: ACM, 2011.

[8] D. Cassou, B. Bertran, N. Loriant, and C. Consel, "A Generative Programming Approach to Developing Pervasive Computing Systems," in *GPCE'09: Proceedings of the 8th International Conference on Generative Programming and Component Engineering*. Denver, CO, USA: ACM Press, 2009, pp. 137–146.

[9] A. R. Perry, "The FlightGear Flight Simulator," in *Proceedings of the USENIX Annual Technical Conference*, 2004.

[10] "Parrot AR.Drone," 2010. [Online]. Available: http://ardrone.parrot.com/

[11] S. Miller, "Specifying the Mode Logic of a Flight Guidance System in CoRE and SCR," in *FMSP'98: Proceedings of the Second Workshop on Formal Methods in Software Practice*. ACM, 1998, pp. 44–53.

[12] J. Windsor and K. Hjortnaes, "Time and Space Partitioning in Spacecraft Avionics," in *SMC-IT'09: Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology*. IEEE, 2009, pp. 13–20.

[13] "ARP-4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment (SAE)," 1996. [Online]. Available: http://standards.sae.org/arp4761

[14] L. Grunske, "Transformational Patterns for the Improvement of Safety Properties in Architectural Specifications," *VikingPLoP'03: Proceedings of the Nordic Conference on Pattern Languages of Programs*, vol. 3, pp. 3–5, 2003.

[15] A. Tribble, D. Lempia, and S. Miller, "Software Safety Analysis of a Flight Guidance System," in *DASC'02: Proceedings of the 21st Digital Avionics Systems Conference*, vol. 2. IEEE, 2002, pp. 13C1–1.

[16] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[17] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering Self-Adaptive Systems through Feedback Loops," in *Software Engineering for Self-Adaptive Systems*, B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 48–70.

[18] "ARINC 664, AFDX: Avionics Full DupleX Switched Ethernet (Aeronautical Radio, Inc.)," 2005. [Online]. Available: http://www.arinc.com/

[19] D. Luckham and J. Vera, "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, 1995.

[20] P. Feiler, "The Architecture Analysis & Design Language (AADL): An Introduction," DTIC Document, Tech. Rep., 2006.

[21] S. Vestal, "An Overview of the Architecture Analysis & Design Language (AADL) Error Model Annex," in *AADL Workshop*, 2005.

[22] P. Dissaux and F. Singhoff, "Stood and Cheddar: AADL as a Pivot Language for Analysing Performances of Real Time Architectures," in *Proceedings of the European Real Time System conference. Toulouse, France*, 2008.

[23] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a Flexible Real Time Scheduling Framework," *ACM SIGAda Ada Letters*, vol. XXIV, pp. 1–8, November 2004.

[24] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, "From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite," *ACM Transactions on Embedded Computing Systems*, vol. 7, pp. 42:1–42:25, August 2008.

[25] S. Burmester, M. Tichy, and H. Giese, "Modeling Reconfigurable Mechatronic Systems with Mechatronic UML," in *Proceedings of Model-Driven Architecture: Foundations and Applications*. Citeseer, 2004.

[26] M. Faugere, T. Bourbeau, R. d. Simone, and S. Gerard, "MARTE: Also an UML Profile for Modeling AADL Applications," in *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 359–364.

[27] B. Dion, "Correct-By-Construction Methods for the Development of Safety-Critical Applications," *SAE transactions*, vol. 113, no. 7, pp. 242–249, 2004.

[28] E. Denney and B. Fischer, "Certifiable Program Generation," in *GPCE'05: Proceedings of the 4th International Conference on Generative Programming and Component Engineering*. Springer, 2005, pp. 17–28.